Capitolo 4

Classi, package e interfacce

di Michael Morrison

IN QUESTO CAPITOLO

Finora si è evitato di discutere della programmazione orientata agli oggetti e di come essa sia correlata a Java. Nel Capitolo 2 in realtà sono stati toccati alcuni argomenti relativi alla programmazione orientata agli oggetti, ma è stata appositamente evitata una discussione completa; lo scopo di questo capitolo è colmare questa lacuna. Si inizia con una breve discussione sulla programmazione orientata agli oggetti in senso generale, dopodiché sono presentati gli elementi specifici del linguaggio Java che forniscono supporto per la programmazione orientata agli oggetti: le classi, i package e le interfacce.

Con questo capitolo termina l'apprendimento di base del linguaggio Java. Le classi sono l'ultimo componente fondamentale del linguaggio che è necessario studiare per diventare validi programmatori. Dopo aver acquisito un'approfondita conoscenza delle classi e del modo in cui funzionano, sarà possibile scrivere veri programmi Java.

Introduzione alla programmazione orientata agli oggetti

Ci si potrebbe chiedere quali siano i vantaggi offerti dagli oggetti e dalla tecnologia orientata agli oggetti. È qualcosa di cui preoccuparsi, e in tal caso, perché? Se si mettono da parte tutte le esagerazioni intorno a questo argomento, si scopre una tecnologia molto potente che offre numerosi benefici per la progettazione di software. Il problema è che i concetti orientati agli oggetti sono difficili da capire e non è possibile sfruttarne i vantaggi se non li si comprende a fondo. Perciò, la completa comprensione della teoria che sta dietro la programmazione orientata agli oggetti si ottiene nel tempo grazie alla pratica.

La grande confusione tra gli sviluppatori in merito alla tecnologia orientata agli oggetti ha causato confusione tra gli utenti di computer in generale. Quanti prodotti dichiarano di essere orientati agli oggetti? Considerato che l'orientamento agli oggetti è un problema relativo alla progettazione del software, che cosa può significare tutto ciò per un consumatore di software? Sotto diversi aspetti, "orientato agli oggetti" è diventato uno slogan. La verità è che il mondo reale è già orientato agli oggetti, cosa che non sorprende nessuno; l'importanza della tecnologia orientata agli oggetti è che permette ai programmatori di progettare software nello stesso modo in cui percepiscono il mondo reale.

Dopo aver chiarito alcuni equivoci legati alla questione dell'orientamento agli oggetti, è possibile metterli da parte e pensare al significato del termine "orientato agli oggetti" per la progettazione di software. Questa introduzione pone le basi per capire come la progettazione orientata agli oggetti renda i programmi più veloci, più semplici e più affidabili. Tutto inizia con gli oggetti. Anche se questo capitolo si concentra principalmente su Java, l'argomento di questo paragrafo in realtà si applica a tutti i linguaggi orientati agli oggetti.

Oggetti

Gli *oggetti* sono "raccolte" di dati e procedure che agiscono sui dati. Le *procedure* sono conosciute anche con il nome di *metodi*. L'unione di dati e metodi costituisce un mezzo per rappresentare nel software gli oggetti del mondo reale, in modo più preciso. Senza gli oggetti, la rappresentazione di un problema del mondo reale nel software richiede un salto logico significativo. D'altra parte, gli oggetti permettono ai programmatori di risolvere i problemi del mondo reale nel dominio del software in modo molto più semplice e logico.

Come indica il nome, gli oggetti sono il fulcro della tecnologia orientata agli oggetti; per capire i benefici che essi offrono nel software, è sufficiente pensare alle caratteristiche comuni di tutti gli oggetti del mondo reale; leoni, automobili e computer hanno tutti due caratteristiche comuni: lo stato e il comportamento. Ad esempio, lo stato di un leone include il colore, il peso e se il leone è stanco o affamato. I leoni hanno anche dei comportamenti: ruggiscono, dormono e cacciano. Lo stato di un'automobile include la velocità corrente, il tipo di trasmissione, la trazione a due o a quattro ruote motrici, le luci accese o spente e la marcia inserita; i comportamenti includono curvare, frenare e accelerare.

Come gli oggetti del mondo reale, anche gli oggetti software hanno in comune lo stato e il comportamento. In termini di programmazione, lo stato di un oggetto è determinato dai suoi dati, il comportamento invece è definito dai suoi metodi. Con questa connessione tra oggetti del mondo reale e oggetti software, si inizia a capire come gli oggetti possano aiutare a colmare il distacco tra il mondo reale e il mondo del software in un computer.

Poiché gli oggetti software hanno come modello gli oggetti del mondo reale, è molto più semplice rappresentare questi ultimi nei programmi orientati agli oggetti. È possibile utilizzare l'oggetto leone per rappresentare un leone vero nel software di uno zoo interattivo. Allo stesso modo, gli oggetti automobile possono essere molto utili in un gioco automobilistico. Tuttavia, non è sempre necessario pensare agli oggetti software come se fossero rappresentazioni di oggetti fisici del mondo reale, in quanto possono essere utili anche come modelli di concetti astratti; ad esempio, un *thread* è un oggetto utilizzato nei sistemi software multithreading per rappresentare un flusso dell'esecuzione del programma. Nel prossimo capitolo vengono fornite ulteriori informazioni sui thread e su come vengono utilizzati in Java.

La Figura 4.1 mostra un oggetto software, con i componenti primari e il modo in cui questi sono correlati; l'oggetto software indica chiaramente i due componenti principali di un oggetto: i dati e i metodi. La figura mostra anche la comunicazione (o accesso) tra i dati e i metodi e come vengono inviati i messaggi ai metodi, con le risposte dall'oggetto. I messaggi e le risposte vengono discussi più avanti in questo capitolo

I dati e i metodi all'interno di un oggetto esprimono tutto ciò che l'oggetto rappresenta (lo stato), assieme a ciò che può fare (il comportamento). Un oggetto software che rappresenta un'automobile del mondo reale può avere variabili (dati) che indicano lo stato attuale dell'automobile (viaggia a 100 km/h, è inserita la quarta e le luci sono accese) e dei metodi che permettono di frenare, di accelerare, di sterzare, di cambiare marcia e di accendere e spegnere le luci. Nella Figura 4.2 è mostrato un oggetto software automobile.

In entrambe le Figure 4.1 e 4.2 vi è una linea che separa i metodi dai dati all'interno dell'oggetto. Questa linea può causare un'interpretazione errata, in quanto i metodi hanno pieno accesso ai dati all'interno di un oggetto. La linea serve a indicare la differenza tra la visibilità dei metodi e dei dati al mondo esterno. In questo senso, la *visibilità* di un oggetto si riferisce alle parti a cui ha accesso un altro oggetto. Poiché in base alle impostazioni predefinite i dati degli oggetti sono invisibili, o inaccessibili, agli altri oggetti, le interazioni tra oggetti devono essere gestite tramite i metodi. Il nascondere i dati all'interno di un oggetto determina l'*incapsulamento*.

Incapsulamento

L'*incapsulamento* è il processo di riunire in un unico pacchetto i dati di un oggetto assieme ai suoi metodi; così si ha il vantaggio di nascondere i dettagli dell'implementazione agli altri oggetti. Ciò significa che la parte interna di un oggetto ha una visibilità più limitata rispetto alla porzione esterna. In questo modo è possibile salvaguardare la parte interna da accessi esterni non desiderati.

Figura 4.1
Un oggetto software.

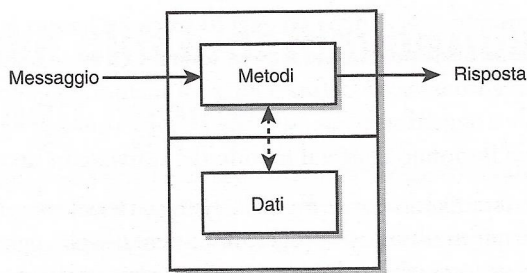
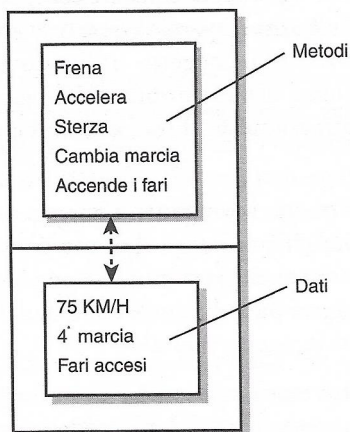



Figura 4.2
Un oggetto software "automobile".

Spesso si fa riferimento alla parte esterna di un oggetto come *interfaccia*, in quanto è la parte che agisce da interfaccia dell'oggetto nei confronti del resto del programma. Poiché gli altri oggetti devono comunicare con l'oggetto in questione solamente tramite la sua interfaccia, la parte interna dell'oggetto è protetta da manomissioni esterne. Inoltre, poiché un programma esterno non ha accesso all'implementazione interna di un oggetto, questa può essere modificata in qualsiasi momento senza influire sulle altre parti del programma.

L'incapsulamento offre ai programmatori due vantaggi principali.

- **Implementazione nascosta.** Si fa riferimento alla protezione dell'implementazione interna di un oggetto. Un oggetto è composto da un'interfaccia pubblica e da una sezione privata che può essere una combinazione di metodi e di dati interni, che costituiscono le parti nascoste dell'oggetto. Il vantaggio principale è che queste sezioni possono essere modificate senza che ciò influisca su altre parti del programma.

- **Modularità.** Significa che un oggetto può essere gestito indipendentemente dagli altri. Poiché il codice sorgente delle sezioni interne di un oggetto viene gestito separatamente dall'interfaccia, si possono apportare modifiche senza che ciò causi problemi alle altre aree. In questo modo è più semplice distribuire gli oggetti all'interno di un sistema.

Messaggi

Un oggetto che agisce da solo raramente è utile; la maggior parte degli oggetti necessita di altri oggetti per fare qualcosa. Ad esempio, l'oggetto automobile da solo senza interazioni sarebbe inutile; aggiungendo un oggetto conducente, la cosa si fa più interessante! Tenendo questo a mente, risulta abbastanza chiaro che gli oggetti necessitano di un meccanismo di comunicazione per interagire tra loro.

Gli oggetti software interagiscono e comunicano tra loro tramite i *messaggi*. Quando l'oggetto conducente vuole che l'oggetto automobile acceleri, gli invia un messaggio. In senso più letterale, è possibile pensare a due persone come se fossero oggetti: se una persona vuole che l'altra si avvicini, le invia un messaggio; per essere più precisi potrebbe dirle: "Per favore, vieni qui". Questo è un messaggio in senso letterale; i messaggi software sono leggermente diversi nella forma, ma non nella teoria: dicono a un oggetto che cosa fare.

Molte volte l'oggetto che riceve il messaggio, per sapere esattamente che cosa fare, necessita di ulteriori informazioni, oltre al messaggio stesso. Quando il conducente dice all'automobile di accelerare, l'automobile deve sapere di quanto. Queste informazioni vengono passate assieme al messaggio come *parametri*.

Da questa discussione si deduce che i messaggi sono composti da tre elementi.

1. L'oggetto che riceve il messaggio (automobile).
2. Il nome dell'azione da eseguire (accelerare).
3. Tutti i parametri necessari per eseguire il metodo (20 Km/h).

Questi tre componenti sono sufficienti a descrivere in modo completo un messaggio per un oggetto. Tutte le interazioni con un oggetto vengono gestite passando un messaggio; ciò significa che gli oggetti in qualunque parte di un sistema possono comunicare con altri oggetti solamente tramite i messaggi.

Per non confondersi, è importante capire che "passare un messaggio" significa "effettuare la chiamata di un metodo". Quando un oggetto invia un messaggio a un altro oggetto, in realtà sta solo richiamando un metodo di quell'oggetto; i parametri del messaggio in realtà sono parametri di un metodo. Nella programmazione orientata agli oggetti, i termini *messaggio* e *metodo* sono sinonimi.

Poiché tutto ciò che può fare un oggetto viene espresso tramite i suoi metodi (interfaccia), il passaggio dei messaggi supporta tutte le possibili interazioni tra gli oggetti. Infatti le interfacce permettono agli oggetti di inviare e di ricevere messaggi, anche se risiedono in posizioni diverse di una rete. Gli oggetti in questo scenario vengono chiamati *oggetti distribuiti*. Java è stato progettato specificatamente per supportare quesit tipi di oggetti.

In realtà, il supporto per gli oggetti distribuiti è un argomento molto complesso e non viene gestito per intero dalla struttura di classi standard di Java. Tuttavia, vi sono nuove estensioni a Java che forniscono il supporto completo per gli oggetti distribuiti.

Classi

In questa discussione sulla programmazione orientata agli oggetti, è stato trattato solamente il concetto di oggetto che già esiste in un sistema. Ci si potrebbe chiedere come gli oggetti entrino a far parte di un sistema. Questa domanda porta alla struttura fondamentale della programmazione orientata agli oggetti: la *classe*. Una classe è un modello o un prototipo che definisce un tipo di oggetto. Una classe è per un oggetto quello che un progetto è per una casa: È possibile costruire molte case sulla base di un unico progetto, che ne definisce la struttura. Le classi funzionano esattamente nello stesso modo, a eccezione del fatto che descrivono la creazione di oggetti.

Nel mondo reale vi sono spesso molti oggetti dello stesso tipo. Utilizzando l'analogia con la casa, nel mondo vi sono molte case diverse, ma tutte hanno delle caratteristiche in comune. In termini orientati agli oggetti, si può dire che una casa è un'*istanza* specifica della classe di oggetti conosciuta come "case". Tutte le case hanno stati e comportamenti comuni che le definiscono come tali. Quando si inizia la costruzione di un isolato di case, di norma si procede sulla base di una serie di progetti. Non sarebbe efficiente creare un progetto per ogni singola casa, in particolare se tra una e l'altra vi sono molte somiglianze. La stessa cosa vale per lo sviluppo di software orientato agli oggetti: perché riscrivere righe e righe di codice, se è possibile riutilizzare il codice che risolve problemi simili?

Nella programmazione orientata agli oggetti, così come nella costruzione di case, è comune avere molti oggetti dello stesso tipo con caratteristiche simili. Come nel caso dei progetti per le case, è possibile creare progetti per gli oggetti che hanno caratteristiche simili. Tutto ciò si può riassumere dicendo che in termini di software le classi sono i progetti per gli oggetti.

Ad esempio, la classe automobile discussa precedentemente conterrebbe diverse variabili che rappresentano lo stato dell'automobile, assieme alle implementazioni dei metodi che permettono al conducente di controllarla. Le variabili di stato dell'automobile rimangono nascoste sotto l'interfaccia. Ogni istanza, o oggetto istanziato, della classe automobile ha una propria serie di variabili di stato. Ciò porta a un altro punto importante: quando si crea un'istanza di un oggetto da una classe, le variabili dichiarate in quella classe vengono allocate in memoria, quindi vengono modificate per mezzo dei metodi dell'oggetto. Le istanze di una stessa classe condividono le implementazioni dei metodi, ma hanno i propri dati.

Se gli oggetti offrono i vantaggi della modularità e di nascondere le informazioni, le classi offrono il vantaggio del riutilizzo. Nello stesso modo in cui i costruttori riutilizzano il progetto di una casa, gli sviluppatori di software possono utilizzare diverse volte una classe per creare molti oggetti. Ognuno di questi oggetti ha i propri dati, ma tutti condividono una singola implementazione dei metodi.

Ereditarietà

Se si desidera che un oggetto sia molto simile a uno che esiste già, ma abbia alcune caratteristiche in più, è sufficiente creare una nuova classe come erede di quella dell'oggetto esistente. L'*ereditarietà* è il processo di creazione di una nuova classe con le caratteristiche di una

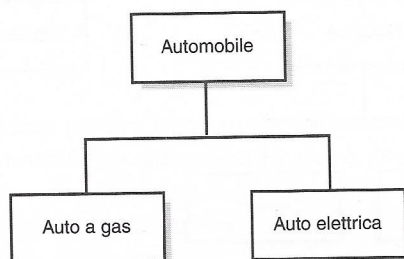
classe esistente e con altre caratteristiche peculiari; costituisce un meccanismo potente e naturale per organizzare e strutturare i programmi.

Finora la discussione sulle classi si è limitata ai dati e ai metodi. In base a quanto appreso, le classi vengono create dal nulla definendone tutti i dati e i metodi associati. L'ereditarietà è uno strumento che permette di creare classi sulla base di altre classi. Quando una classe si basa su un'altra classe, eredita tutte le proprietà di quella, inclusi i dati e i metodi. La classe che eredita viene definita *sottoclasse* (o *classe figlia*), mentre la classe che fornisce le informazioni è detta *superclasse* (o *classe genitore*).

Utilizzando l'esempio dell'automobile, dalla classe automobile è possibile ereditare classi figlie per le automobili a gas e per le automobili elettriche. Entrambe le nuove classi hanno in comune le caratteristiche delle automobili, ma hanno anche alcune caratteristiche proprie. Ad esempio, la classe delle automobili a gas avrebbe un serbatoio e un tappo, l'automobile elettrica avrebbe una batteria e una presa per la ricarica. Ogni sottoclasse eredita dalla superclasse le informazioni sullo stato (sotto forma di dichiarazioni di variabile). La Figura 4.3 mostra la classe genitore automobile e le classi figlie automobile a gas e automobile elettrica.

Ereditare solamente lo stato e i comportamenti di una superclasse non sarebbe granché per una sottoclasse. La vera forza sta nella capacità di ereditare proprietà e metodi e di aggiungerne di nuovi: alle sottoclassi è possibile aggiungere variabili e metodi, oltre a quelli ereditati dalla superclasse (l'automobile elettrica ha *aggiunto* una batteria e una presa per la ricarica). Inoltre, le sottoclassi hanno la capacità di ridefinire i metodi ereditati e di fornire implementazioni diverse per essi. Ad esempio, l'automobile a gas probabilmente sarebbe in grado di andare molto più veloce rispetto all'automobile elettrica. Il metodo "accelerare" per l'automobile a gas potrebbe riflettere questa differenza.

L'ereditarietà delle classi è stata progettata per permettere la maggior flessibilità possibile. Un gruppo di classi correlate è chiamato *struttura ereditaria* o *gerarchia di classe*. Una struttura ereditaria assomiglia a un albero genealogico e, come questo, mostra i rapporti tra le classi. A differenza dell'albero genealogico, le classi in una struttura ereditaria diventano sempre più specifiche a mano a mano che ci si sposta verso il basso. È possibile creare qualsiasi tipo di struttura di classe, anche se è importante non creare troppi livelli, perché in questo modo risulterebbe difficile vedere il rapporto tra le classi. Le classi di automobili indicate nella Figura 4.3 offrono un buon esempio di struttura ereditaria.

Figura 4.3
Oggetti automobile ereditati.

Se si capisce il concetto di ereditarietà, si comprende come le sottoclassi possano avere dati e metodi specializzati, oltre a quelli comuni forniti dalla superclasse. Questo permette ai programmatori di riutilizzare diverse volte il codice nella superclasse, evitando di creare ulteriore codice aumentando la probabilità di errori.

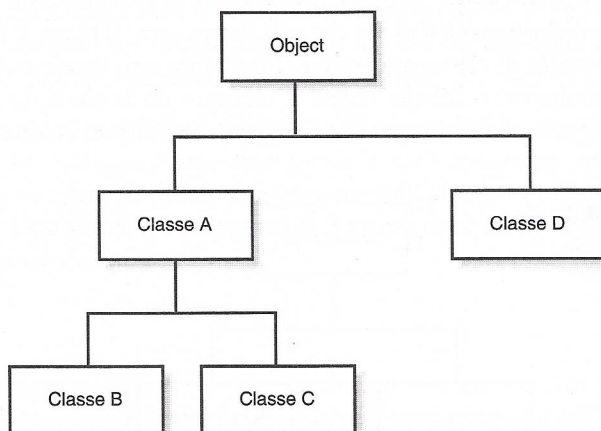
Un ultimo punto relativo all'ereditarietà: è possibile e a volte utile creare superclassi che agiscono solamente da modelli per sottoclassi più facilmente utilizzabili. In questo caso, la superclasse serve solo come astrazione per le funzionalità comuni condivise dalle sottoclassi. Per questo motivo, si parla di *classi astratte*. Non è possibile creare istanze dalle classi astratte, vale a dire che da esse non è possibile creare oggetti. Il motivo è che le classi astratte contengono parti che sono state appositamente lasciate senza implementazione. Più specificamente, queste parti sono composte da metodi che devono ancora essere implementati: i *metodi astratti*.

Tornando all'esempio dell'automobile, il metodo "accelerare" non può essere definito finché non si conoscono le capacità di accelerazione dell'automobile. Naturalmente, l'accelerazione dipende dal tipo di motore; poiché il tipo di motore non è conosciuto nella superclasse, il metodo "accelerare" può essere definito, ma viene lasciato senza implementazione, cosa che rende sia il metodo sia la superclasse automobile astratti. Le classi figlie automobile a gas e automobile elettrica implementano il metodo "accelerare" in base alla capacità dei rispettivi motori.

La gerarchia di classi di Java

In Java, tutte le classi sono sottoclassi di una superclasse chiamata `Object`, la cui gerarchia è illustrata nella Figura 4.4.

Come si può vedere, tutte le classi derivano dalla classe di base `Object`, che in Java è la superclasse di tutte le classi derivate, incluse quelle che compongono l'API di Java.

Figura 4.4
Classi derivate dalla superclasse Object.

Dichiarazione di classi

La sintassi per dichiarare le classi in Java è:

```
class Identificatore {
  CorpoClasse
}
```

Identificatore specifica il nome della nuova classe che, in base alle impostazioni predefinite, deriva da `Object`. Le parentesi graffe racchiudono il corpo della classe, *CorpoClasse*. Ad esempio, si osservi la dichiarazione della classe `Alieno`, che potrebbe essere utilizzata in un gioco spaziale:

```
class Alieno {
  Color colore;
  int   energia;
  int   aggressivita;
}
```

Lo stato dell'oggetto `Alieno` viene definito da tre variabili istanza che rappresentano il colore, l'energia e l'aggressività dell'alieno. Si noti che la classe `Alieno` deriva da `Object`. Per ora la classe `Alieno` non è molto utile e necessita di metodi. La sintassi più semplice per dichiarare i metodi per una classe è:

```
TipoReturn Identificatore(Parametri) {
  CorpoMetodo
}
```

TipoReturn specifica il tipo di dati che il metodo restituisce, *Identificatore* specifica il nome del metodo e *Parametri* specifica i parametri del metodo, se presenti. Come per il corpo delle classi, il corpo di un metodo, *CorpoMetodo*, è racchiuso tra parentesi graffe. In termini di progettazione orientata agli oggetti, metodo è sinonimo di messaggio e il tipo restituito è la risposta dell'oggetto al messaggio. Di seguito viene presentata la dichiarazione per il metodo `morph()`, che potrebbe essere utile alla classe `Alieno` per permettere agli alieni di cambiare forma:

```
void morph(int aggressivita) {
  if (aggressivita < 10) {
    // diventa più piccolo
  }
  else if (aggressivita < 20) {
    // assume una taglia media
  }
  else {
    // diventa un gigante
  }
}
```

Al metodo `morph()` viene passato come unico parametro un intero, `aggressivita`; questo valore viene quindi utilizzato per determinare le nuove dimensioni dell'alieno dopo la metamorfosi. Come si può vedere, l'alieno modifica le dimensioni diventando più grande o più piccolo in base al suo grado di aggressività.

Se si fa in modo che il metodo `morph()` faccia parte della classe `Alieno`, risulta subito evidente che il parametro `aggressivita` non è necessario, in quanto `aggressivita` è già una variabile membro di `Alieno`, a cui hanno accesso tutti i metodi della classe. Ecco la classe `Alieno` con il metodo `morph()`:

```
class Alieno {
  Color colore;
  int   energia;
  int   aggressivita;

  void morph() {
    if (aggressivita < 10) {
      // diventa più piccolo
    }
    else if (aggressivita < 20) {
      // assume una taglia media
    }
    else {
      // diventa un gigante
    }
  }
}
```

Derivazione di classi

Finora la discussione sulla dichiarazione delle classi si è limitata alla creazione di nuove classi derivate da `Object`. Non è opportuno derivare tutte le classi da `Object`, in quanto sarebbe necessario ridefinire i dati e i metodi per ognuna. Per derivare una classe da classi diverse da `Object`, si utilizza la parola chiave `extends`. La sintassi è:

```
class Identificatore extends SuperClasse {
  CorpoClasse
}
```

`Identificatore` indica il nome della nuova classe derivata, *`SuperClasse`* indica il nome della classe genitore e *`CorpoClasse`* è il corpo della nuova classe.

Utilizzando la classe `Alieno` introdotta in precedenza come base per un esempio di derivazione, che cosa succederebbe se si avesse una classe `Nemico` che definisse informazioni generali utili per tutti i nemici? Senza dubbio si vorrebbe far derivare la classe `Alieno` dalla classe `Nemico`, in modo da sfruttare le funzionalità standard fornite da quest'ultima. Di seguito viene presentata la classe `Alieno` derivata dalla classe `Nemico` utilizzando la parola chiave `extends`:

```
class Alieno extends Nemico {
  Color colore;
  int   energia;
  int   aggressivita;

  void morph() {
    if (aggressivita < 10) {
      // diventa più piccolo
    }
```

```
      else if (aggressivita < 20) {
        // assume una taglia media
      }
      else {
        // diventa un gigante
      }
    }
}
```

In questo caso si assume che la dichiarazione della classe `Nemico` sia già disponibile nello stesso package di `Alieno`. In realtà, è possibile creare classi derivate anche da classi esterne, importando prima la superclasse con l'istruzione `import`.

I package vengono discussi più avanti in questo capitolo. Per il momento è possibile pensare a un package come a un gruppo di classi correlate.

Se fosse necessario importare la classe `Nemico`, l'istruzione sarebbe:

```
import Nemico;
```

Ridefinizione di metodi

A volte è utile *ridefinire* i metodi nelle classi derivate. Ad esempio, se la classe `Nemico` avesse un metodo `move()`, si potrebbe variare il movimento in base al tipo di nemico. Alcuni tipi di nemici possono volare con motivi specifici, altri possono strisciare in modo casuale. Per permettere alla classe `Alieno` di avere il proprio movimento, è possibile ridefinire il metodo `move()` con una versione specifica per il movimento dell'alieno. La classe `Nemico` sarebbe:

```
class Nemico {
...
  void move() {
    // muove il nemico
  }
}
```

Allo stesso modo, la classe `Alieno` con il metodo `move()` ridefinito sarebbe:

```
class Alieno {
  Color colore;
  int   energia;
  int   aggressivita;
  void move() {
    // muove l'alieno
  }
  void morph() {
    if (aggressivita < 10) {
      // diventa più piccolo
    }
    else if (aggressivita < 20) {
      // assume una taglia media
    }
    else {
      // diventa un gigante
    }
  }
}
```

Quando si crea un'istanza della classe `Alieno` e si richiama il metodo `move()`, viene eseguito il nuovo metodo `move()` in `Alieno` e non il metodo originale `move()` ridefinito in `Nemico`. La ridefinizione dei metodi costituisce un utilizzo semplice ma potente della progettazione orientata agli oggetti.

Overloading di metodi

Un'altra potente tecnica orientata agli oggetti è l'*overloading dei metodi*, che permette di specificare diversi tipi di informazioni (parametri) da inviare a un metodo. Per eseguire l'overloading di un metodo, se ne dichiara una nuova versione con lo stesso nome, ma con parametri diversi.

Ad esempio, il metodo `move()` per la classe `Alieno` potrebbe avere due versioni diverse: una per il movimento generale e una per spostarsi a un punto specifico. La versione generale è quella già definita che sposta l'alieno in base al suo stato attuale. La dichiarazione di questa versione è:

```
void move() {
  // muove l'alieno
}
```

Per permettere all'alieno di spostarsi in un punto specifico, si esegue l'overloading del metodo `move()` con una versione che utilizza i parametri `x` e `y`, che specificano il punto in cui spostarsi. La nuova versione di `move()` è:

```
void move(int x, int y) {
  // muove l'alieno nella posizione x,y
}
```

Si noti che l'unica differenza tra i due metodi è data dai parametri: il primo metodo `move()` non ha parametri, il secondo ne ha due interi.

Ci si potrebbe chiedere in che modo il compilatore determini quale metodo viene richiamato in un programma, se entrambi hanno lo stesso nome. Il compilatore mantiene i parametri di ogni metodo assieme al nome; quando in un programma viene effettuata una chiamata a un metodo, il compilatore controlla il nome e i parametri per determinare quale metodo viene chiamato.

In questo caso, le chiamate ai metodi `move()` sono facilmente distinguibili in base all'assenza o alla presenza dei parametri `int`.

Modificatori di accesso

L'accesso alle variabili e ai metodi nelle classi Java viene effettuato tramite i *modificatori di accesso*, che definiscono diversi livelli di accesso tra i membri delle classi e il mondo esterno (gli altri oggetti). I modificatori di accesso vengono dichiarati immediatamente prima del tipo di una variabile membro o del tipo restituito da un metodo. Esistono quattro modificatori di accesso: quello predefinito, `public`, `protected` e `private`.

I modificatori di accesso influiscono non solo sulla visibilità dei membri di una classe, ma anche delle classi stesse. Tuttavia, la visibilità delle classi è strettamente collegata ai package, che vengono discussi più avanti in questo capitolo.

Il modificatore di accesso predefinito

Il modificatore di accesso predefinito specifica che solo le classi all'interno dello stesso package hanno accesso alle variabili e ai metodi di una classe. I membri di una classe con accesso predefinito hanno visibilità limitata alle altre classi all'interno dello stesso package. Non esiste una parola chiave per dichiarare il modificatore di accesso predefinito, che viene applicato automaticamente in assenza di altri modificatori di accesso. Ad esempio, i membri della classe `Alieno` hanno tutti accesso predefinito, in quanto non è stato specificato alcun modificatore di accesso. Di seguito vengono riportati gli esempi di una variabile e di un metodo con accesso predefinito:

```
long lunghezza;
void getLunghezza() {
  return lunghezza;
}
```

Si noti che né la variabile membro né il metodo includono un modificatore di accesso e pertanto assumono implicitamente il modificatore di accesso predefinito.

Il modificatore di accesso public

Il modificatore di accesso `public` specifica che le variabili e i metodi di una classe sono accessibili a chiunque, sia all'interno sia all'esterno della classe. Ciò significa che i membri `public` di una classe hanno visibilità globale e che qualsiasi altro oggetto può accedervi. Di seguito vengono riportati alcuni esempi di variabili membro `public`:

```
public int conta;
public boolean seiAttivo;
```

Il modificatore di accesso protected

Il modificatore di accesso `protected` specifica che i membri di una classe sono accessibili solo ai metodi di quella classe e delle relative sottoclassi. Ciò significa che i membri `protected` di una classe hanno visibilità limitata alle sottoclassi. Di seguito vengono riportati gli esempi di una variabile e di un metodo `protected`:

```
protected char iniziale;
protected char ottieniIniziale() {
  return iniziale;
}
```

Il modificatore di accesso private

Il modificatore di accesso `private` è il più restrittivo e specifica che i membri di una classe sono accessibili solo alla classe in cui sono definiti. Ciò significa che nessun'altra classe,

incluse le sottoclassi, ha accesso ai membri private di una classe. Di seguito vengono riportati alcuni esempi di variabili membro private:

```
private String nome;
private double quantoSeiGrande;
```

Il modificatore static

A volte è necessario avere una variabile o un metodo comune per tutti gli oggetti di una particolare classe; a questo scopo, si utilizza il modificatore static.

Di norma, per ogni istanza di una classe vengono allocate nuove variabili. Quando una variabile viene dichiarata static, viene allocata una sola volta, indipendentemente da quanti oggetti vengono istanziati. Il risultato è che tutti gli oggetti istanziati condividono la stessa istanza della variabile static. Allo stesso modo, l'implementazione di un metodo static è esattamente uguale per tutti gli oggetti di una particolare classe. Ciò significa che i metodi static hanno accesso solo a variabili static.

Di seguito vengono presentati gli esempi di una variabile membro e di un metodo static:

```
static int contaRif;
static int getContaRif() {
  return contaRif;
}
```

Un effetto collaterale positivo dei membri static è che è possibile accedervi senza dover creare un'istanza di una classe. Per il metodo System.out.println() utilizzato nel capitolo precedente non è stato istanziato nessun oggetto System.out è una variabile membro static della classe System, vale a dire che vi si può accedere senza dover istanziare un oggetto System.

Il modificatore final

Un altro modificatore utile relativo al controllo dell'utilizzo dei membri delle classi è final, che specifica che una variabile ha un valore costante o che un metodo non può essere ridefinito in una sottoclasse. Come si può dedurre dal nome, il modificatore final indica che il membro di una classe è la versione finale autorizzata per la stessa.

Di seguito sono riportati alcuni esempi di variabili membro final:

```
final public int numDollari = 25;
final boolean braccioRotto = false;
```

A chi proviene dal mondo del C++, le variabili final potrebbero sembrare familiari, poiché sono simili alle variabili const del C++, infatti devono sempre essere inizializzate al momento della dichiarazione e successivamente il loro valore non può più essere modificato.

Il modificatore synchronized

Il modificatore synchronized viene utilizzato per specificare che un metodo è *a thread protetto*, vale a dire che in un metodo synchronized è concesso un solo percorso di esecuzione

alla volta. In un ambiente multithreading come quello di Java, è possibile che nello stesso codice vengano eseguiti contemporaneamente diversi percorsi di esecuzione. Il modificatore `synchronized` modifica questa regola, permettendo a un solo thread di accedere a un metodo in un dato momento e obbligando gli altri thread ad aspettare il loro turno. I thread e percorsi di esecuzione sono discussi dettagliatamente nel prossimo capitolo.

Il modificatore native

Il modificatore `native` viene utilizzato per identificare i metodi con implementazione nativa. Esso informa il compilatore di Java che l'implementazione di un metodo si trova in un file esterno C. Per questo motivo le dichiarazioni dei metodi `native` sono diverse da quelle degli altri metodi di Java: non hanno corpo. Ecco la dichiarazione di un metodo `native`:

```
native int calcTotale();
```

Si noti che la dichiarazione del metodo termina semplicemente con un punto e virgola, senza parentesi graffe che racchiudono il codice Java. Questo perché i metodi nativi vengono implementati nel codice C, che risiede in file sorgente C esterni.

Classi e metodi astratti

Nell'introduzione alla programmazione orientata agli oggetti all'inizio di questo capitolo si è parlato di classi e di metodi astratti. Per ricapitolare, una *classe astratta* è una classe che viene implementata parzialmente e il cui scopo è solo quello di facilitare la progettazione. Le classi astratte sono composte da uno o più *metodi astratti*, che sono metodi dichiarati, ma lasciati senza corpo (non implementati).

La classe `Nemico` discussa precedentemente in questo capitolo è un candidato ideale a diventare una classe astratta. In realtà non è opportuno creare un oggetto `Nemico`, che sarebbe troppo generale. Tuttavia, la classe `Nemico` ha uno scopo logico come superclasse per classi di nemici più specifiche, ad esempio la classe `Alieno`. Per trasformare la classe `Nemico` in una classe astratta, si utilizza la parola chiave `abstract`, come indicato di seguito:

```
abstract class Nemico {
  abstract void move();
  abstract void move(int x, int y);
}
```

La parola chiave `abstract` viene utilizzata prima della dichiarazione di classe per `Nemico`; in questo modo si indica al compilatore che la classe `Nemico` è astratta. Si noti che entrambi i metodi `move()` vengono dichiarati come astratti. Poiché non è chiaro come si muove un nemico generico, i metodi `move()` in `Nemico` sono stati lasciati senza implementazione (astratti).

Vi sono alcune limitazioni all'utilizzo di `abstract`, di cui è opportuno essere a conoscenza. Innanzitutto, non è possibile rendere astratti i costruttori, di cui si discute nel prossimo paragrafo che tratta la creazione di oggetti. Secondariamente, non è possibile rendere astratti i metodi statici.

Questa limitazione deriva dal fatto che i metodi statici vengono dichiarati per tutte le classi, pertanto non è possibile fornire un'implementazione derivata per un metodo statico astratto. Infine, non è permesso rendere astratti i metodi privati. Questa limitazione potrebbe sembrare eccessiva, ma quando si deriva una classe da una superclasse con metodi astratti, è necessario ridefinire e implementare tutti i metodi astratti, altrimenti non sarà possibile istanziare la nuova classe, che rimarrà anch'essa astratta. Poiché le classi derivate non vedono i membri privati delle loro superclassi, inclusi i metodi, non sarebbe possibile ridefinire e implementare i metodi astratti dalla superclasse, e quindi non sarebbe possibile implementare classi (non astratte). Se si potessero derivare solo classi astratte, non sarebbe possibile farci granché.

Casting

Nonostante il casting (o *conversione*) tra diversi tipi di dati sia stato discusso nel Capitolo 2, l'introduzione delle classi permette di affrontare nuovamente questo argomento. Il casting tra le classi può essere suddiviso in tre diverse situazioni.

- Casting da una sottoclasse a una superclasse.
- Casting da una superclasse a una sottoclasse.
- Casting tra sottoclassi.

Nel caso del casting da una sottoclasse a una superclasse, è possibile eseguire la conversione implicitamente o esplicitamente. *Casting implicito* significa semplicemente che non si fa nulla, *casting esplicito* significa che è necessario fornire il tipo di classe tra parentesi, come si fa per i tipi di dati primitivi. Il casting da una sottoclasse a una superclasse è del tutto affidabile, in quanto le sottoclassi contengono le informazioni che le collegano alle rispettive superclassi. Quando si esegue il casting da una superclasse a una sottoclasse, è necessario effettuare la conversione in modo esplicito. Questa operazione non è completamente affidabile, in quanto il compilatore non ha modo di sapere se la classe che viene convertita è una sottoclasse della superclasse interessata. Infine, il casting tra sottoclassi in Java non è permesso. Per capire meglio l'argomento, si controlli il seguente codice:

```
Double d1 = new Double(5.238);
Number n = d1;
Double d2 = (Double)n;
Long l = d1;  // questo non funziona!
```

In questo esempio, vengono creati e assegnati l'uno all'altro oggetti involucro dei tipi di dati, che vengono descritti nel Capitolo 9. Per ora, tutto ciò che è necessario sapere è che le sottoclassi `Double` e `Long` derivano entrambe dalla classe `Number`. In questo esempio, dopo essere stato creato, l'oggetto `Double d1` viene assegnato a un oggetto `Number`. Questo è un esempio di casting implicito da una sottoclasse a una superclasse, del tutto lecito. Successivamente, a un altro oggetto `Double d2` viene assegnato il valore dell'oggetto `Number`. Questa volta è necessario un casting esplicito, in quanto si sta effettuando la conversione da una superclasse a una sottoclasse, operazione non necessariamente affidabile. Infine, a un oggetto `Long` viene assegnato il valore di un oggetto `Double`: questo è un casting tra sottoclassi, che in Java non è permesso e che causa un errore di compilazione.

Creazione di oggetti

Benché nella programmazione orientata agli oggetti la maggior parte del lavoro di progettazione consista nel creare le classi, in realtà non si ottengono benefici da questo lavoro finché da queste classi non si creano istanze (oggetti). Per utilizzare una classe in un programma, è necessario crearne prima un'istanza.

Il costruttore

Prima di entrare nel dettaglio di come si crea un oggetto, è necessario conoscere un metodo importante: il *costruttore*. Quando si crea un oggetto, di norma se ne inizializzano le variabili membro. Il costruttore è un metodo speciale che è possibile implementare in qualsiasi classe e che permette di inizializzare le variabili e di eseguire qualsiasi altra operazione quando da una classe viene creato un oggetto. Al costruttore viene sempre attribuito lo stesso nome della classe. Il Listato 4.1 contiene il codice sorgente completo per la classe `Alieno`, che contiene due costruttori.

Listato 4.1 *La classe Alieno.*

```
class Alieno extends Nemico {
  protected Color colore;
  protected int   energia;
  protected int   aggressivita;

  public Alieno() {
    colore = Color.green;
    energia = 100;
    aggressivita = 15;
  }

  public Alieno(Color c, int e, int a) {
    colore = c;
    energia = e;
    aggressivita = a;
  }

  public void move() {
    // muove l'alieno
  }

  public void move(int x, int y) {
    // muove l'alieno nella posizione x,y
  }

  public void morph() {
    if (aggressivita < 10) {
      // diventa più piccolo
    }
    else if (aggressivita < 20) {
      // assume una taglia media
    }
```

```
        else {
          // diventa un gigante
        }
      }
    }
```

La classe Alieno utilizza l'overloading dei metodi per fornire due diversi costruttori: il primo non ha parametri e inizializza le variabili membro in base ai valori predefiniti, il secondo specifica il colore, l'energia e l'aggressività dell'alieno e inizializza le variabili membro in base a questi valori. Oltre a contenere i nuovi costruttori, questa versione di Alieno utilizza i modificatori di accesso per assegnare esplicitamente i livelli di accesso a ogni metodo e a ogni variabile membro. Si tratta di una buona pratica a cui conviene abituarsi.

Questa versione della classe Alieno si trova nel file sorgente Enemy1.java nel CD-ROM allegato al libro. Il file con il codice sorgente di Enemy1.Java include anche la classe Nemico. Queste classi sono solo degli esempi con poche funzionalità, ma rappresentano buoni esempi di progettazione di classi in Java e possono essere compilate in classi Java.

L'operatore new

Per creare un'istanza di una classe, si dichiara una variabile oggetto e si utilizza l'operatore new. Quando si lavora sugli oggetti, una dichiarazione indica semplicemente quale tipo di oggetto deve rappresentare una variabile. L'oggetto non viene effettivamente creato finché non viene utilizzato l'operatore new. Di seguito sono presentati due esempi che utilizzano questo operatore per creare istanze della classe Alieno:

```
Alieno unAlieno = new Alieno();
Alieno unaltroAlieno;
unaltroAlieno = new Alieno(Color.red, 56, 24);
```

Nel primo esempio, viene dichiarata la variabile unAlieno e viene creato l'oggetto utilizzando l'operatore new con un assegnamento direttamente nella dichiarazione. Nel secondo esempio, viene prima dichiarata la variabile unaltroAlieno, quindi l'oggetto viene creato e assegnato in un enunciato separato.

Chi ha esperienza con il C++ senza dubbio conosce l'operatore new. *Questo operatore in Java funziona in modo simile alla sua controparte in C++, ma è importante tenere a mente che è necessario utilizzarlo sempre per creare oggetti in Java, a differenza della versione del C++ che viene utilizzata solo quando si lavora con i puntatori agli oggetti. Poiché Java non supporta i puntatori, per creare nuovi oggetti deve essere sempre utilizzato l'operatore* new.

Distruzione di oggetti

Quando un oggetto esce dal proprio ambito, viene rimosso dalla memoria o eliminato. In modo simile al costruttore, che viene richiamato quando viene creato un oggetto, Java permette di definire un *distruttore*, che viene richiamato quando viene eliminato un oggetto.

A differenza del costruttore, che ha lo stesso nome della classe, il distruttore è chiamato finalize(). Il metodo finalize() permette di eseguire routine correlate all'eliminazione dell'oggetto e viene definito nel seguente modo:

```
void finalize() {
   // codice di pulizia
}
```

Non è detto che il metodo finalize() venga richiamato da Java non appena un oggetto esce dal suo ambito, poiché Java elimina gli oggetti tramite il sistema di *garbage collection*, che viene eseguito a intervalli irregolari. Poiché un oggetto non viene realmente eliminato finché non è stata eseguita la garbage collection, fino a quel momento non viene neanche richiamato il metodo finalize(). Sapendo ciò, non ci si dovrebbe basare sul metodo finalize() per operazioni in cui il tempo è un fattore critico. In termini generali, si può dire che è necessario inserire il codice nel metodo finalize() solo raramente, in quanto il sistema di esecuzione di Java effettua da solo un buon lavoro di eliminazione degli oggetti.

Package

Java fornisce uno strumento potente per raggruppare classi e interfacce correlate in un'unica unità: i package. Le interfacce vengono discusse più avanti in questo capitolo; per quanto riguarda i *package*, in parole semplici si tratta di gruppi di classi e di interfacce correlate. Essi costituiscono un valido meccanismo per gestire un grosso gruppo di classi e interfacce, evitando potenziali conflitti tra i nomi. L'API stessa di Java è implementata come un gruppo di package.

Ad esempio, le classi Alieno e Nemico sviluppate precedentemente in questo capitolo potrebbero essere parte di un package Nemico, assieme a qualsiasi altro oggetto nemico. Riunendo le classi in un package, si permette loro di beneficiare del modificatore di accesso predefinito, che fornisce alle classi l'accesso alle informazioni delle altre classi nello stesso package.

Dichiarazione di package

La sintassi dell'istruzione package è:

```
package Identificatore;
```

Questa istruzione deve essere inserita all'inizio di un'unità di compilazione (un singolo file sorgente), prima delle dichiarazioni delle classi. Ogni classe che si trova in un'unità di compilazione con un'istruzione package è considerata parte di quel package. È possibile suddividere le classi tra diverse unità di compilazione, assicurandosi però di includere in ognuna un'istruzione package.

I package possono essere annidati all'interno di altri package. In questo caso, l'interprete di Java si aspetta che la struttura delle directory che contiene le classi eseguibili corrisponda alla gerarchia dei package.

Importazione di package

Quando si devono utilizzare delle classi all'esterno del package in cui si sta lavorando, è necessario utilizzare l'istruzione `import`, che permette di importare in un'unità di compilazione classi che fanno parte di altri package. È possibile importare contemporaneamente singole classi o package interi. La sintassi dell'enunciato `import` è:

```
import Identificatore;
```

Identificatore è il nome della classe o del package di classi da importare. Tornando all'esempio della classe `Alieno`, la variabile membro `colore` è un'istanza dell'oggetto `Color`, che fa parte della libreria di classi AWT (Abstract Windowing Toolkit) di Java. Affinché il compilatore possa capire questo tipo di variabile membro, è necessario importare la classe `Color`, utilizzando una delle due istruzioni seguenti:

```
import java.awt.Color;
import java.awt.*;
```

La prima istruzione importa la classe specifica `Color`, che si trova nel package `java.awt`, la seconda importa tutte le classi del package `java.awt`. Si noti che la seguente istruzione non funziona:

```
import java.*;
```

Questo perché non è possibile importare package annidati con la specifica *. Questo carattere jolly funziona solo quando si importano tutte le classi di un determinato package.

Esiste un altro modo per importare oggetti da altri package: il *riferimento esplicito al package*. Facendo riferimento esplicitamente al nome del package ogni volta che si utilizza un oggetto, si può evitare l'utilizzo dell'istruzione `import`. Utilizzando questa tecnica, la dichiarazione della variabile membro `colore` in `Alieno` sarebbe:

```
java.awt.Color colore;
```

Normalmente non è necessario fare riferimento esplicitamente al nome del package per una classe esterna, cosa che allunga il nome della classe e rende più difficile la lettura del codice. L'eccezione a questa regola si ha quando due package contengono classi con lo stesso nome. In questo caso, assieme al nome delle classi è necessario utilizzare esplicitamente il nome del package.

Visibilità delle classi

Precedentemente in questo capitolo sono stati discussi i modificatori di accesso, che influiscono sulla visibilità delle classi e dei membri delle classi. Poiché la visibilità dei membri delle classi è determinata in relazione alle classi, ci si può chiedere cosa significhi visibilità per una classe: la visibilità delle classi è determinata in relazione ai package.

Ad esempio, una classe `public` è visibile alle classi di altri package. In realtà, `public` è l'unico modificatore di accesso utilizzabile per le classi; senza di esso, le classi assumo le impostazioni predefinite rimanendo visibili alle altre classi del package, ma invisibili alle classi esterne.

Classi interne

La maggior parte delle classi Java è definita a livello di package, vale a dire che ogni classe è membro di un particolare package. Se per una classe non si indica esplicitamente l'associazione a un package specifico, viene assunto il package predefinito. Le classi definite a livello di package sono dette *classi di livello superiore*. Prima di Java 1.1, le classi di livello superiore erano le sole supportate. Java 1.1 ha introdotto un approccio più aperto alla definizione delle classi, supportando le *classi interne*, che possono essere definite in qualsiasi ambito. Ciò significa che una classe può essere definita come membro di un'altra classe, all'interno di un blocco di istruzioni o in modo anonimo con un'espressione.

Nonostante possano sembrare un miglioramento secondario al linguaggio Java, le classi interne in realtà rappresentano un'importante modifica: si consideri che rappresentano l'unica modifica apportata al linguaggio Java con la versione 1.1. Gli altri miglioramenti introdotti riguardano le nuove API. Perché preoccuparsi di cambiare il linguaggio per qualcosa che sembra così astratto come le classi interne? La risposta a questa domanda non è semplice. Anziché addentrarsi in una discussione che va oltre l'ambito di questo capitolo, si può riassumere l'esigenza delle classi interne dicendo che il nuovo modello di eventi AWT di Java 1.1 per funzionare in modo appropriato necessitava specificatamente di un meccanismo come quello fornito dalle classi interne.

Le regole che governano l'ambito di una classe interna corrispondono esattamente a quelle che governano le variabili. Il nome di una classe interna non è visibile all'esterno del suo ambito, salvo che venga specificato per intero, cosa che aiuta a strutturare le classi all'interno di un package. Il codice di una classe interna può utilizzare nomi semplici per tutto ciò che si trova all'interno dell'ambito in cui è racchiusa, incluse le classi e le variabili membro delle classi di tale ambito e anche le variabili locali di blocchi che la comprendono. Inoltre, è possibile definire una classe di livello superiore come membro statico di un'altra classe di livello superiore. A differenza delle classi interne, le classi di livello superiore non possono utilizzare direttamente le variabili istanza di un'altra classe. La possibilità di annidare le classi in questo modo permette a qualsiasi classe di livello superiore di fornire per un gruppo secondario di classi di livello superiore correlate in modo logico un'organizzazione sullo stile di quella dei package.

Di seguito è presentato un esempio semplice di classe interna:

```
public class Esterna {
  int x, y;

  public int calcArea() {
    return x * y;
  }
  class Interna {
    int z;
    public int calcVolume() {
      return calcArea() * z;
    }
  }
}
```

In questo esempio, una classe interna chiamata `Interna` viene dichiarata all'interno di una classe chiamata `Esterna`. Come si può vedere, la dichiarazione della classe interna assomiglia alla dichiarazione di una classe normale (esterna). In realtà questo esempio non è molto utile, ma dà un'idea di come sono strutturate le classi interne.

Il supporto per le classi interne in Java 1.1 è stato fornito interamente dal compilatore di Java e non ha richiesto nessun cambiamento alla macchina virtuale. Questo è uno dei motivi principali per cui gli architetti di Java hanno voluto modificare il linguaggio in modo che supportasse le classi interne, in quanto sapevano che ciò non avrebbe avuto alcun impatto sulla macchina virtuale.

Interfacce

L'ultimo argomento di questa discussione sulla programmazione orientata agli oggetti in Java riguarda le interfacce. Un'*interfaccia* è un prototipo per una classe ed è utile dal punto di vista della progettazione logica.

Precedentemente in questo capitolo si è appreso che una classe astratta è una classe che è stata lasciata parzialmente senza implementazione, in quanto utilizza metodi astratti che sono essi stessi non implementati. Le interfacce sono classi astratte lasciate completamente senza implementazione, il che in questo caso significa che nelle classi non è stato implementato nessun metodo. Inoltre, i dati membro delle interfacce sono limitati alle variabili statiche finali, cioè sono costanti.

I vantaggi dell'utilizzo delle interfacce sono gli stessi offerti dall'utilizzo delle classi astratte. Le interfacce costituiscono un mezzo per definire i protocolli per una classe, senza preoccuparsi dei dettagli dell'implementazione. Questo beneficio, apparentemente poco importante, può rendere molto più semplice la gestione di progetti di grandi dimensioni; dopo aver progettato le interfacce, lo sviluppo delle classi può avvenire senza doversi preoccupare della comunicazione tra di esse.

Un altro aspetto importante delle interfacce è la possibilità per una classe di implementarne diverse. Questo rappresenta una svolta per il concetto di ereditarietà multipla, supportata in C++ ma non in Java. L'*ereditarietà multipla* permette di derivare una classe da diverse classi genitore. Nonostante sia uno strumento potente, l'ereditarietà multipla è una funzionalità difficile e complessa del C++, di cui i progettisti di Java hanno deciso di fare a meno. La loro soluzione è stata quella di permettere alle classi di Java di implementare diverse interfacce.

La differenza principale tra l'ereditarietà di diverse interfacce e la vera ereditarietà multipla è che la prima permette di ereditare solo le descrizioni dei metodi, non le implementazioni. Se una classe implementa diverse interfacce, deve fornire tutte le funzionalità per i metodi definiti in queste ultime. Nonostante ponga sicuramente limiti maggiori rispetto all'ereditarietà multipla, questo approccio è comunque molto utile. Questa è la funzionalità delle interfacce che le differenzia dalle classi astratte.

Dichiarazione di interfacce

La sintassi per creare le interfacce è:

```
interface Identificatore {
  CorpoInterfaccia
}
```

Identificatore è il nome dell'interfaccia, mentre *CorpoInterfaccia* fa riferimento ai metodi astratti e alle variabili statiche finali che compongono l'interfaccia. Poiché si assume che tutti i metodi in un'interfaccia siano astratti, non è necessario utilizzare la parola chiave `abstract`.

Implementazione di interfacce

Poiché un'interfaccia è un prototipo, o modello, di una classe, per arrivare a una classe utilizzabile è necessario implementare un'interfaccia. L'implementazione di un'interfaccia è simile alla derivazione da una classe, a eccezione del fatto che è necessario implementare tutti i metodi definiti nell'interfaccia. A tale scopo si utilizza la parola chiave `implements`. La sintassi è:

```
class Identificatore implements Interfaccia {
  CorpoClasse
}
```

Identificatore indica il nome della nuova classe, *Interfaccia* è il nome dell'interfaccia da implementare e *CorpoClasse* è il corpo della nuova classe. Nel Listato 4.2 è riportato il codice sorgente di `Enemy2.java`, che contiene una versione come interfaccia di `Nemico`, assieme alla classe `Alieno` che implementa l'interfaccia.

Listato 4.2 *L'interfaccia Nemico e la classe Alieno.*

```
package Nemico;

import java.awt.Color;

interface Nemico {
  abstract public void move();
  abstract public void move(int x, int y);
}

class Alieno implements Nemico {
  protected Color colore;
  protected int   energia;
  protected int   aggressivita;

  public Alieno() {
    colore = Color.green;
    energia = 100;
    aggressivita = 15;
  }
```

```
    public Alieno(Color c, int e, int a) {
      colore = c;
      energia = e;
      aggressivita = a;
    }

    public void move() {
      // muove l'alieno
    }

    public void move(int x, int y) {
      // muove l'alieno nella posizione x,y
    }

    public void morph() {
      if (aggressivita < 10) {
        // diventa più piccolo
      }
      else if (aggressivita < 20) {
        // assume una taglia media
      }
      else {
        // diventa un gigante
      }
    }
}
```

Riepilogo

In questo capitolo sono stati discussi i concetti fondamentali della programmazione orientata agli oggetti e anche i costrutti specifici di Java che permettono di mettere in pratica i concetti orientati agli oggetti: le classi, i package e le interfacce. Sono stati spiegati i vantaggi che si traggono dall'utilizzo delle classi e come implementare oggetti da queste ultime. Sono stati descritti anche i messaggi (metodi), che costituiscono il meccanismo di comunicazione tra gli oggetti. Si è appreso come l'ereditarietà costituisca uno strumento potente per riutilizzare il codice e per creare progetti modulari, come i package permettano di raggruppare in modo logico gruppi di classi simili, rendendo più semplice la gestione di serie di classi di grandi dimensioni. Infine, si è visto come le interfacce forniscono un modello da cui derivare nuove classi in modo strutturale.

Si è ora pronti a passare a funzionalità più avanzate del linguaggio Java, quali i thread e il multithreading, che vengono discussi nel prossimo capitolo.